File: APPNOTE.TXT - .ZIP File Format Specification
Version: 5.2 - NOTIFICATION OF CHANGE
Revised: 06/02/2003

Disclaimer

Although PKWARE will attempt to supply current and accurate information relating to its file formats, algorithms, and the subject programs, the possibility of error or omission can not be eliminated. PKWARE therefore expressly disclaims any warranty that the information contained in the associated materials relating to the subject programs and/or the format of the files created or accessed by the subject programs and/or the algorithms used by the subject programs, or any other matter, is current, correct or accurate as delivered. Any risk of damage due to any possible inaccurate information is assumed by the user of the information. Furthermore, the information relating to the subject programs and/or the file formats created or accessed by the subject programs and/or the algorithms used by the subject programs is subject to change without notice.

If the version of this file is marked as a NOTIFICATION OF CHANGE, the content defines an Early Feature Specification (EFS) change to the .ZIP file format that may be subject to modification prior to publication of the Final Feature Specification (FFS). This document may also contain information on Planned Feature Specifications (PFS) defining recognized future extensions.

General Format of a .ZIP file

Files stored in arbitrary order. Large .ZIP files can span multiple diskette media or be split into user-defined segment sizes.

Overall .ZIP file format:

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

A. Local file header:

```
local file header signature     4 bytes  (0x04034b50)
version needed to extract       2 bytes
general purpose bit flag        2 bytes
compression method              2 bytes
last mod file time              2 bytes
last mod file date              2 bytes
```

```
      crc-32                          4 bytes
      compressed size                 4 bytes
      uncompressed size               4 bytes
      file name length                2 bytes
      extra field length              2 bytes

      file name (variable size)
      extra field (variable size)
```

B. File data

Immediately following the local header for a file is the compressed or stored data for the file. The series of [local file header][file data][data descriptor] repeats for each file in the .ZIP archive.

C. Data descriptor:

```
      crc-32                          4 bytes
      compressed size                 4 bytes
      uncompressed size               4 bytes
```

This descriptor exists only if bit 3 of the general purpose bit flag is set (see below). It is byte aligned and immediately follows the last byte of compressed data. This descriptor is used only when it was not possible to seek in the output .ZIP file, e.g., when the output .ZIP file was standard output or a non seekable device. For Zip64 format archives, the compressed and uncompressed sizes are 8 bytes each.

D. Central directory structure:

```
    [file header 1]
    .
    .
    .
    [file header n]
    [digital signature]
```

File header:

```
      central file header signature   4 bytes  (0x02014b50)
      version made by                 2 bytes
      version needed to extract       2 bytes
      general purpose bit flag        2 bytes
      compression method              2 bytes
      last mod file time              2 bytes
      last mod file date              2 bytes
      crc-32                          4 bytes
      compressed size                 4 bytes
      uncompressed size               4 bytes
      file name length                2 bytes
      extra field length              2 bytes
      file comment length             2 bytes
      disk number start               2 bytes
      internal file attributes        2 bytes
      external file attributes        4 bytes
      relative offset of local header 4 bytes

      file name (variable size)
      extra field (variable size)
      file comment (variable size)
```

```
    Digital signature:

      header signature                4 bytes  (0x05054b50)
      size of data                    2 bytes
      signature data (variable size)

E.  Zip64 end of central directory record

      zip64 end of central dir
      signature                       4 bytes  (0x06064b50)
      size of zip64 end of central
      directory record                8 bytes
      version made by                 2 bytes
      version needed to extract       2 bytes
      number of this disk             4 bytes
      number of the disk with the
      start of the central directory  4 bytes
      total number of entries in the
      central directory on this disk  8 bytes
      total number of entries in the
      central directory               8 bytes
      size of the central directory   8 bytes
      offset of start of central
      directory with respect to
      the starting disk number        8 bytes
      zip64 extensible data sector    (variable size)

F.  Zip64 end of central directory locator

      zip64 end of central dir locator
      signature                       4 bytes  (0x07064b50)
      number of the disk with the
      start of the zip64 end of
      central directory               4 bytes
      relative offset of the zip64
      end of central directory record 8 bytes
      total number of disks           4 bytes

G.  End of central directory record:

      end of central dir signature    4 bytes  (0x06054b50)
      number of this disk             2 bytes
      number of the disk with the
      start of the central directory  2 bytes
      total number of entries in the
      central directory on this disk  2 bytes
      total number of entries in
      the central directory           2 bytes
      size of the central directory   4 bytes
      offset of start of central
      directory with respect to
      the starting disk number        4 bytes
      .ZIP file comment length        2 bytes
      .ZIP file comment       (variable size)

H.  Explanation of fields:

    version made by (2 bytes)

        The upper byte indicates the compatibility of the file
        attribute information.  If the external file attributes
        are compatible with MS-DOS and can be read by PKZIP for
        DOS version 2.04g then this value will be zero.  If these
```

```
    attributes are not compatible, then this value will
    identify the host system on which the attributes are
    compatible.  Software can use this information to determine
    the line record format for text files etc.  The current
    mappings are:

    0 - MS-DOS and OS/2 (FAT / VFAT / FAT32 file systems)
    1 - Amiga                     2 - OpenVMS
    3 - Unix                      4 - VM/CMS
    5 - Atari ST                  6 - OS/2 H.P.F.S.
    7 - Macintosh                 8 - Z-System
    9 - CP/M                     10 - Windows NTFS
   11 - MVS (OS/390 - Z/OS)      12 - VSE
   13 - Acorn Risc               14 - VFAT
   15 - alternate MVS            16 - BeOS
   17 - Tandem                   18 - OS/400
   19 thru 255 - unused

    The lower byte indicates the version number of the
    software used to encode the file.  The value/10
    indicates the major version number, and the value
    mod 10 is the minor version number.

version needed to extract (2 bytes)

    The minimum software version needed to extract the
    file, mapped as above.  For Zip64 format archives,
    this value should not be less than 45.

general purpose bit flag: (2 bytes)

    Bit 0: If set, indicates that the file is encrypted.

    (For Method 6 - Imploding)
    Bit 1: If the compression method used was type 6,
           Imploding, then this bit, if set, indicates
           an 8K sliding dictionary was used.  If clear,
           then a 4K sliding dictionary was used.
    Bit 2: If the compression method used was type 6,
           Imploding, then this bit, if set, indicates
           3 Shannon-Fano trees were used to encode the
           sliding dictionary output.  If clear, then 2
           Shannon-Fano trees were used.

    (For Methods 8 and 9 - Deflating)
    Bit 2  Bit 1
      0      0    Normal (-en) compression option was used.
      0      1    Maximum (-exx/-ex) compression option was used.
      1      0    Fast (-ef) compression option was used.
      1      1    Super Fast (-es) compression option was used.

    Note:  Bits 1 and 2 are undefined if the compression
           method is any other.

    Bit 3: If this bit is set, the fields crc-32, compressed
           size and uncompressed size are set to zero in the
           local header.  The correct values are put in the
           data descriptor immediately following the compressed
           data.  (Note: PKZIP version 2.04g for DOS only
           recognizes this bit for method 8 compression, newer
           versions of PKZIP recognize this bit for any
           compression method.)
```

Bit 4: Reserved for use with method 8, for enhanced deflating.

Bit 5: If this bit is set, this indicates that the file is compressed patched data. (Note: Requires PKZIP version 2.70 or greater)

Bit 6: Strong encryption. If this bit is set, you should set the version needed to extract value to at least 50 and you must also set bit 0. If AES encryption is used, the version needed to extract value must be at least 51.

Bit 7: Currently unused.

Bit 8: Currently unused.

Bit 9: Currently unused.

Bit 10: Currently unused.

Bit 11: Currently unused.

Bit 12: Reserved by PKWARE for enhanced compression.

Bit 13: Reserved by PKWARE.

Bit 14: Reserved by PKWARE.

Bit 15: Reserved by PKWARE.

compression method: (2 bytes)

(see accompanying documentation for algorithm descriptions)

```
 0 - The file is stored (no compression)
 1 - The file is Shrunk
 2 - The file is Reduced with compression factor 1
 3 - The file is Reduced with compression factor 2
 4 - The file is Reduced with compression factor 3
 5 - The file is Reduced with compression factor 4
 6 - The file is Imploded
 7 - Reserved for Tokenizing compression algorithm
 8 - The file is Deflated
 9 - Enhanced Deflating using Deflate64(tm)
10 - PKWARE Data Compression Library Imploding
11 - Reserved by PKWARE
12 - File is compressed using BZIP2 algorithm
```

date and time fields: (2 bytes each)

The date and time are encoded in standard MS-DOS format. If input came from standard input, the date and time are those at which compression was started for this data.

CRC-32: (4 bytes)

The CRC-32 algorithm was generously contributed by David Schwaderer and can be found in his excellent book "C Programmers Guide to NetBIOS" published by Howard W. Sams & Co. Inc. The 'magic number' for the CRC is 0xdebb20e3. The proper CRC pre and post

```
        conditioning is used, meaning that the CRC register
        is pre-conditioned with all ones (a starting value
        of 0xffffffff) and the value is post-conditioned by
        taking the one's complement of the CRC residual.
        If bit 3 of the general purpose flag is set, this
        field is set to zero in the local header and the correct
        value is put in the data descriptor and in the central
        directory.

    compressed size: (4 bytes)
    uncompressed size: (4 bytes)

        The size of the file compressed and uncompressed,
        respectively.  If bit 3 of the general purpose bit flag
        is set, these fields are set to zero in the local header
        and the correct values are put in the data descriptor and
        in the central directory.  If an archive is in zip64 format
        and the value in this field is 0xFFFFFFFF, the size will be
        in the corresponding 8 byte zip64 extended information
        extra field.

    file name length: (2 bytes)
    extra field length: (2 bytes)
    file comment length: (2 bytes)

        The length of the file name, extra field, and comment
        fields respectively.  The combined length of any
        directory record and these three fields should not
        generally exceed 65,535 bytes.  If input came from standard
        input, the file name length is set to zero.

    disk number start: (2 bytes)

        The number of the disk on which this file begins.  If an
        archive is in zip64 format and the value in this field is
        0xFFFF, the size will be in the corresponding 4 byte zip64
        extended information extra field.

    internal file attributes: (2 bytes)

        The lowest bit of this field indicates, if set, that
        the file is apparently an ASCII or text file.  If not
        set, that the file apparently contains binary data.
        The remaining bits are unused in version 1.0.

        Bits 1 and 2 are reserved for use by PKWARE.

    external file attributes: (4 bytes)

        The mapping of the external attributes is
        host-system dependent (see 'version made by').  For
        MS-DOS, the low order byte is the MS-DOS directory
        attribute byte.  If input came from standard input, this
        field is set to zero.

    relative offset of local header: (4 bytes)

        This is the offset from the start of the first disk on
        which this file appears, to where the local header should
        be found.  If an archive is in zip64 format and the value
        in this field is 0xFFFFFFFF, the size will be in the
        corresponding 8 byte zip64 extended information extra field.
```

file name: (Variable)

The name of the file, with optional relative path. The path stored should not contain a drive or device letter, or a leading slash. All slashes should be forward slashes '/' as opposed to backwards slashes '\' for compatibility with Amiga and Unix file systems etc. If input came from standard input, there is no file name field.

extra field: (Variable)

This is for expansion. If additional information needs to be stored for special needs or for specific platforms, it should be stored here. Earlier versions of the software can then safely skip this file, and find the next file or header. This field will be 0 length in version 1.0.

In order to allow different programs and different types of information to be stored in the 'extra' field in .ZIP files, the following structure should be used for all programs storing data in this field:

header1+data1 + header2+data2 . . .

Each header should consist of:

```
  Header ID - 2 bytes
  Data Size - 2 bytes
```

Note: all fields stored in Intel low-byte/high-byte order.

The Header ID field indicates the type of data that is in the following data block.

Header ID's of 0 thru 31 are reserved for use by PKWARE. The remaining ID's can be used by third party vendors for proprietary usage.

The current Header ID mappings defined by PKWARE are:

```
0x0001        ZIP64 extended information extra field
0x0007        AV Info
0x0008        Reserved for future Unicode file name data (PFS)
0x0009        OS/2
0x000a        NTFS
0x000c        OpenVMS
0x000d        Unix
0x000f        Patch Descriptor
0x0014        PKCS#7 Store for X.509 Certificates
0x0015        X.509 Certificate ID and Signature for
              individual file
0x0016        X.509 Certificate ID for Central Directory
0x0017        Strong Encryption Header
0x0018        Record Management Controls
0x0065        IBM S/390 (Z390), AS/400 (I400) attributes
              - uncompressed
0x0066        IBM S/390 (Z390), AS/400 (I400) attributes
              - compressed
```

Third party mappings commonly used are:

```
0x2605        ZipIt Macintosh
0x2705        ZipIt Macintosh 1.3.5+
0x07c8        Macintosh
0x2805        ZipIt Macintosh 1.3.5+
0x334d        Info-ZIP Macintosh
0x4341        Acorn/SparkFS
0x4453        Windows NT security descriptor (binary ACL)
0x4704        VM/CMS
0x470f        MVS
0x4b46        FWKCS MD5 (see below)
0x4c41        OS/2 access control list (text ACL)
0x4d49        Info-ZIP OpenVMS
0x4f4c        Xceed original location extra field
0x5356        AOS/VS (ACL)
0x5455        extended timestamp
0x554e        Xceed unicode extra field
0x5855        Info-ZIP Unix (original, also OS/2, NT, etc)
0x6542        BeOS/BeBox
0x756e        ASi Unix
0x7855        Info-ZIP Unix (new)
0xfd4a        SMS/QDOS
```

Detailed descriptions of Extra Fields defined by third party mappings will be documented as information on these data structures is made available to PKWARE. PKWARE does not guarantee the accuracy of any published third party data.

The Data Size field indicates the size of the following data block. Programs can use this value to skip to the next header block, passing over any data blocks that are not of interest.

Note: As stated above, the size of the entire .ZIP file header, including the file name, comment, and extra field should not exceed 64K in size.

In case two different programs should appropriate the same Header ID value, it is strongly recommended that each program place a unique signature of at least two bytes in size (and preferably 4 bytes or bigger) at the start of each data area. Every program should verify that its unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

-OS/2 Extra Field:

The following is the layout of the OS/2 attributes "extra" block. (Last Revision 09/05/95)

Note: all fields stored in Intel low-byte/high-byte order.

```
          Value         Size          Description
          -----         ----          -----------
  (OS/2)  0x0009        2 bytes       Tag for this "extra" block type
          TSize         2 bytes       Size for the following data block
          BSize         4 bytes       Uncompressed Block Size
          CType         2 bytes       Compression type
          EACRC         4 bytes       CRC value for uncompress block
          (var)         variable      Compressed block
```

The OS/2 extended attribute structure (FEA2LIST) is compressed and then stored in it's entirety within this structure. There will only ever be one "block" of data in VarFields[].

-UNIX Extra Field:

The following is the layout of the Unix "extra" block. Note: all fields are stored in Intel low-byte/high-byte order.

```
        Value         Size          Description
        -----         ----          -----------
(UNIX)  0x000d        2 bytes       Tag for this "extra" block type
        TSize         2 bytes       Size for the following data block
        Atime         4 bytes       File last access time
        Mtime         4 bytes       File last modification time
        Uid           2 bytes       File user ID
        Gid           2 bytes       File group ID
        (var)         variable      Variable length data field
```

The variable length data field will contain file type specific data. Currently the only values allowed are the original "linked to" file names for hard or symbolic links, and the major and minor device node numbers for character and block device nodes. Since device nodes cannot be either symbolic or hard links, only one set of variable length data is stored. Link files will have the name of the original file stored. This name is NOT NULL terminated. Its size can be determined by checking TSize - 12. Device entries will have eight bytes stored as two 4 byte entries (in little endian format). The first entry will be the major device number, and the second the minor device number.

-OpenVMS Extra Field:

The following is the layout of the OpenVMS attributes "extra" block.

Note: all fields stored in Intel low-byte/high-byte order.

```
        Value         Size        Description
        -----         ----        -----------
(VMS)   0x000c        2 bytes     Tag for this "extra" block type
        TSize         2 bytes     Size of the total "extra" block
        CRC           4 bytes     32-bit CRC for remainder of the block
        Tag1          2 bytes     OpenVMS attribute tag value #1
        Size1         2 bytes     Size of attribute #1, in bytes
        (var.)        Size1       Attribute #1 data
        .
        .
        .
        TagN          2 bytes     OpenVMS attribute tage value #N
        SizeN         2 bytes     Size of attribute #N, in bytes
        (var.)        SizeN       Attribute #N data
```

Rules:

1. There will be one or more of attributes present, which will each be preceded by the above TagX & SizeX values. These values are identical to the ATR$C_XXXX and

```
     ATR$S_XXXX constants which are defined in ATR.H under
     OpenVMS C.  Neither of these values will ever be zero.

  2. No word alignment or padding is performed.

  3. A well-behaved PKZIP/OpenVMS program should never produce
     more than one sub-block with the same TagX value.  Also,
     there will never be more than one "extra" block of type
     0x000c in a particular directory record.

 -NTFS Extra Field:

  The following is the layout of the NTFS attributes
  "extra" block. (Note: At this time the Mtime, Atime
  and Ctime values may be used on any WIN32 system.)

  Note: all fields stored in Intel low-byte/high-byte order.

        Value      Size       Description
        -----      ----       -----------
(NTFS)  0x000a     2 bytes    Tag for this "extra" block type
        TSize      2 bytes    Size of the total "extra" block
        Reserved   4 bytes    Reserved for future use
        Tag1       2 bytes    NTFS attribute tag value #1
        Size1      2 bytes    Size of attribute #1, in bytes
        (var.)     Size1      Attribute #1 data
        .
        .
        .
        TagN       2 bytes    NTFS attribute tag value #N
        SizeN      2 bytes    Size of attribute #N, in bytes
        (var.)     SizeN      Attribute #N data

  For NTFS, values for Tag1 through TagN are as follows:
  (currently only one set of attributes is defined for NTFS)

        Tag        Size       Description
        -----      ----       -----------
        0x0001     2 bytes    Tag for attribute #1
        Size1      2 bytes    Size of attribute #1, in bytes
        Mtime      8 bytes    File last modification time
        Atime      8 bytes    File last access time
        Ctime      8 bytes    File creation time

 -PATCH Descriptor Extra Field:

  The following is the layout of the Patch Descriptor "extra"
  block.

  Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size     Description
        -----     ----     -----------
(Patch) 0x000f    2 bytes  Tag for this "extra" block type
        TSize     2 bytes  Size of the total "extra" block
        Version   2 bytes  Version of the descriptor
        Flags     4 bytes  Actions and reactions (see below)
        OldSize   4 bytes  Size of the file about to be patched
        OldCRC    4 bytes  32-bit CRC of the file to be patched
        NewSize   4 bytes  Size of the resulting file
        NewCRC    4 bytes  32-bit CRC of the resulting file

  Actions and reactions
```

```
        Bits          Description
        ----          ----------------
        0             Use for autodetection
        1             Treat as selfpatch
        2-3           RESERVED
        4-5           Action (see below)
        6-7           RESERVED
        8-9           Reaction (see below) to absent file
        10-11         Reaction (see below) to newer file
        12-13         Reaction (see below) to unknown file
        14-15         RESERVED
        16-31         RESERVED

        Actions

        Action       Value
        ------       -----
        none         0
        add          1
        delete       2
        patch        3

        Reactions

        Reaction     Value
        --------     -----
        ask          0
        skip         1
        ignore       2
        fail         3

        Patch support is provided by PKPatchMaker(tm) technology and is
        covered under U.S. Patents and Patents Pending.

       -PKCS#7 Store for X.509 Certificates:

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size       Description
        -----     ----       -----------
(Store) 0x0014    2 bytes    Tag for this "extra" block type
        TSize     2 bytes    Size of the store data
        (var)     TSize      Data about the store


       -X.509 Certificate ID and Signature for individual file:

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size       Description
        -----     ----       -----------
(CID)   0x0015    2 bytes    Tag for this "extra" block type
        TSize     2 bytes    Size of data that follows
        (var)     TSize      Data

       -X.509 Certificate ID and Signature for central directory:

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size       Description
        -----     ----       -----------
(CDID)  0x0016    2 bytes    Tag for this "extra" block type
```

```
        TSize      2 bytes  Size of data that follows
        (var)      TSize    Data

       -Strong Encryption Header (EFS):

        Value      Size     Description
        -----      ----     -----------
        0x0017     2 bytes  Tag for this "extra" block type
        TSize      2 bytes  Size of data that follows
        Format     2 bytes  Format definition for this record
        AlgID      2 bytes  Encryption algorithm identifier
        Bitlen     2 bytes  Bit length of encryption key
        Flags      2 bytes  Processing flags
        (var)      TSize    Reserved for future certificate data


       -Record Management Controls:

        Value      Size     Description
        -----      ----     -----------
(Rec-CTL) 0x0018   2 bytes  Tag for this "extra" block type
        CSize      2 bytes  Size of total extra block data
        Tag1       2 bytes  Record control attribute 1
        Size1      2 bytes  Size of attribute 1, in bytes
        Data       Size1    Attribute 1 data
          .
          .
          .
        TagN       2 bytes  Record control attribute N
        SizeN      2 bytes  Size of attribute N, in bytes
        Data       SizeN    Attribute N data

       -MVS Extra Field:

        The following is the layout of the MVS "extra" block.
        Note: Some fields are stored in Big Endian format.
        All text is in EBCDIC format unless otherwise specified.

        Value        Size          Description
        -----        ----          -----------
  (MVS) 0x0065       2 bytes       Tag for this "extra" block type
        TSize        2 bytes       Size for the following data block
        ID           4 bytes       EBCDIC "Z390" 0xE9F3F9F0 or
                                   "T4MV" for TargetFour
        (var)        TSize-4       Attribute data


       -OS/400 Extra Field:

        The following is the layout of the OS/400 "extra" block.
        Note: Some fields are stored in Big Endian format.
        All text is in EBCDIC format unless otherwise specified.

        Value        Size          Description
        -----        ----          -----------
(OS400) 0x0065       2 bytes       Tag for this "extra" block type
        TSize        2 bytes       Size for the following data block
        ID           4 bytes       EBCDIC "I400" 0xC9F4F0F0 or
                                   "T4MV" for TargetFour
        (var)        TSize-4       Attribute data


       -ZipIt Macintosh Extra Field (long):
```

The following is the layout of the ZipIt extra block for Macintosh. The local-header and central-header versions are identical. This block must be present if the file is stored MacBinary-encoded and it should not be used if the file is not stored MacBinary-encoded.

	Value	Size	Description
(Mac2)	0x2605	Short	tag for this extra block type
	TSize	Short	total data size for this block
	"ZPIT"	beLong	extra-field signature
	FnLen	Byte	length of FileName
	FileName	variable	full Macintosh filename
	FileType	Byte[4]	four-byte Mac file type string
	Creator	Byte[4]	four-byte Mac creator string

-ZipIt Macintosh Extra Field (short, for files):

The following is the layout of a shortened variant of the ZipIt extra block for Macintosh (without "full name" entry). This variant is used by ZipIt 1.3.5 and newer for entries of files (not directories) that do not have a MacBinary encoded file. The local-header and central-header versions are identical.

	Value	Size	Description
(Mac2b)	0x2705	Short	tag for this extra block type
	TSize	Short	total data size for this block (12)
	"ZPIT"	beLong	extra-field signature
	FileType	Byte[4]	four-byte Mac file type string
	Creator	Byte[4]	four-byte Mac creator string
	fdFlags	beShort	attributes from FInfo.frFlags, may be omitted
	0x0000	beShort	reserved, may be omitted

-ZipIt Macintosh Extra Field (short, for directories):

The following is the layout of a shortened variant of the ZipIt extra block for Macintosh used only for directory entries. This variant is used by ZipIt 1.3.5 and newer to save some optional Mac-specific information about directories. The local-header and central-header versions are identical.

	Value	Size	Description
(Mac2c)	0x2805	Short	tag for this extra block type
	TSize	Short	total data size for this block (12)
	"ZPIT"	beLong	extra-field signature
	frFlags	beShort	attributes from DInfo.frFlags, may be omitted
	View	beShort	ZipIt view flag, may be omitted

The View field specifies ZipIt-internal settings as follows:

Bits of the Flags:

bit 0	if set, the folder is shown expanded (open) when the archive contents are viewed in ZipIt.
bits 1-15	reserved, zero;

-ZIP64 Extended Information Extra Field:

The following is the layout of the ZIP64 extended information "extra" block. If one of the size or offset fields in the Local or Central directory record is too small to hold the required data, a ZIP64 extended information record is created. The order of the fields in the ZIP64 extended information record is fixed, but the fields will only appear if the corresponding Local or Central directory record field is set to 0xFFFF or 0xFFFFFFFF.

Note: all fields stored in Intel low-byte/high-byte order.

```
        Value      Size       Description
        -----      ----       -----------
(ZIP64) 0x0001     2 bytes    Tag for this "extra" block type
        Size       2 bytes    Size of this "extra" block
        Original
        Size       8 bytes    Original uncompresseed file size
        Compressed
        Size       8 bytes    Size of compressed data
        Relative Header
        Offset     8 bytes    Offset of local header record
        Disk Start
        Number     4 bytes    Number of the disk on which
                              this file starts
```

This entry in the Local header must include BOTH original and compressed file sizes.

-FWKCS MD5 Extra Field:

The FWKCS Contents_Signature System, used in automatically identifying files independent of file name, optionally adds and uses an extra field to support the rapid creation of an enhanced contents_signature:

```
    Header ID = 0x4b46
    Data Size = 0x0013
    Preface   = 'M','D','5'
    followed by 16 bytes containing the uncompressed file's
    128_bit MD5 hash(1), low byte first.
```

When FWKCS revises a .ZIP file central directory to add this extra field for a file, it also replaces the central directory entry for that file's uncompressed file length with a measured value.

FWKCS provides an option to strip this extra field, if present, from a .ZIP file central directory. In adding this extra field, FWKCS preserves .ZIP file Authenticity Verification; if stripping this extra field, FWKCS preserves all versions of AV through PKZIP version 2.04g.

FWKCS, and FWKCS Contents_Signature System, are trademarks of Frederick W. Kantor.

(1) R. Rivest, RFC1321.TXT, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. ll.76-77: "The MD5 algorithm is being placed in the public domain for review and possible adoption as a

standard."

file comment: (Variable)

The comment for this file.

number of this disk: (2 bytes)

The number of this disk, which contains central directory end record. If an archive is in zip64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 end of central directory field.

number of the disk with the start of the central directory: (2 bytes)

The number of the disk on which the central directory starts. If an archive is in zip64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 end of central directory field.

total number of entries in the central dir on this disk: (2 bytes)

The number of central directory entries on this disk. If an archive is in zip64 format and the value in this field is 0xFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

total number of entries in the central dir: (2 bytes)

The total number of files in the .ZIP file. If an archive is in zip64 format and the value in this field is 0xFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

size of the central directory: (4 bytes)

The size (in bytes) of the entire central directory. If an archive is in zip64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

offset of start of central directory with respect to the starting disk number: (4 bytes)

Offset of the start of the central directory on the disk on which the central directory starts. If an archive is in zip64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

.ZIP file comment length: (2 bytes)

The length of the comment for this .ZIP file.

.ZIP file comment: (Variable)

The comment for this .ZIP file.

zip64 extensible data sector   (variable size)

(currently reserved for use by PKWARE)

I. General notes:

1) All fields unless otherwise noted are unsigned and stored in Intel low-byte:high-byte, low-word:high-word order.

2) String fields are not null terminated, since the length is given explicitly.

3) Local headers should not span disk boundaries. Also, even though the central directory can span disk boundaries, no single record in the central directory should be split across disks.

4) The entries in the central directory may not necessarily be in the same order that files appear in the .ZIP file.

5) Spanned/Split archives created using PKZIP for Windows (V2.50 or greater), PKZIP Command Line (V2.50 or greater), or PKZIP Explorer will include a special spanning signature as the first 4 bytes of the first segment of the archive. This signature (0x08074b50) will be followed immediately by the local header signature for the first file in the archive. A special spanning marker may also appear in spanned/split archives if the spanning or splitting process starts but only requires one segement. In this case the 0x08074b50 signature will be replaced with the temporary spanning marker signature of 0x30304b50. Spanned/split archives created with this special signature are compatible with all versions of PKZIP from PKWARE. Split archives can only be uncompressed by other versions of PKZIP that know how to create a split archive.

6) If one of the fields in the end of central directory record is too small to hold required data, the field should be set to -1 (0xFFFF or 0xFFFFFFFF) and the Zip64 format record should be created.

7) The end of central directory record and the Zip64 end of central directory locator record must reside on the same disk when splitting or spanning an archive.

UnShrinking - Method 1

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm with partial clearing. The initial code size is 9 bits, and the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations in several respects:

1) The code size is controlled by the compressor, and is not automatically increased when codes larger than the current code size are created (but not necessarily used). When

the decompressor encounters the code sequence 256 (decimal) followed by 1, it should increase the code size read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the sequence 256,1 is encountered.

2) When the table becomes full, total clearing is not performed. Rather, when the compressor emits the code sequence 256,2 (decimal), the decompressor should clear all leaf nodes from the Ziv-Lempel tree, and continue to use the current code size. The nodes that are cleared from the Ziv-Lempel tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence 256,2 at any time.

Expanding - Methods 2-5

The Reducing algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences, and the second algorithm takes the compressed stream from the first algorithm and applies a probabilistic compression method.

The probabilistic compression stores an array of 'follower sets' S(j), for j=0 to 255, corresponding to each possible ASCII character. Each set contains between 0 and 32 characters, to be denoted as S(j)[0],...,S(j)[m], where m<32. The sets are stored at the beginning of the data area for a Reduced file, in reverse order, with S(255) first, and S(0) last.

The sets are encoded as { N(j), S(j)[0],...,S(j)[N(j)-1] }, where N(j) is the size of set S(j). N(j) can be 0, in which case the follower set for S(j) is empty. Each N(j) value is encoded in 6 bits, followed by N(j) eight bit character values corresponding to S(j)[0] to S(j)[N(j)-1] respectively. If N(j) is 0, then no values for S(j) are stored, and the value for N(j-1) immediately follows.

Immediately after the follower sets, is the compressed data stream. The compressed data stream can be interpreted for the probabilistic decompression as follows:

```
let Last-Character <- 0.
loop until done
    if the follower set S(Last-Character) is empty then
        read 8 bits from the input stream, and copy this
        value to the output stream.
    otherwise if the follower set S(Last-Character) is non-empty then
        read 1 bit from the input stream.
        if this bit is not zero then
            read 8 bits from the input stream, and copy this
            value to the output stream.
        otherwise if this bit is zero then
            read B(N(Last-Character)) bits from the input
            stream, and assign this value to I.
            Copy the value of S(Last-Character)[I] to the
```

```
            output stream.

    assign the last value placed on the output stream to
    Last-Character.
end loop

B(N(j)) is defined as the minimal number of bits required to
encode the value N(j)-1.

The decompressed stream from above can then be expanded to
re-create the original file as follows:

let State <- 0.

loop until done
    read 8 bits from the input stream into C.
    case State of
        0:  if C is not equal to DLE (144 decimal) then
                copy C to the output stream.
            otherwise if C is equal to DLE then
                let State <- 1.

        1:  if C is non-zero then
                let V <- C.
                let Len <- L(V)
                let State <- F(Len).
            otherwise if C is zero then
                copy the value 144 (decimal) to the output stream.
                let State <- 0

        2:  let Len <- Len + C
            let State <- 3.

        3:  move backwards D(V,C) bytes in the output stream
            (if this position is before the start of the output
            stream, then assume that all the data before the
            start of the output stream is filled with zeros).
            copy Len+3 bytes from this position to the output stream.
            let State <- 0.
    end case
end loop

The functions F,L, and D are dependent on the 'compression
factor', 1 through 4, and are defined as follows:

For compression factor 1:
    L(X) equals the lower 7 bits of X.
    F(X) equals 2 if X equals 127 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 1 bit of X) * 256 + Y + 1.
For compression factor 2:
    L(X) equals the lower 6 bits of X.
    F(X) equals 2 if X equals 63 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 2 bits of X) * 256 + Y + 1.
For compression factor 3:
    L(X) equals the lower 5 bits of X.
    F(X) equals 2 if X equals 31 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 3 bits of X) * 256 + Y + 1.
For compression factor 4:
    L(X) equals the lower 4 bits of X.
    F(X) equals 2 if X equals 15 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 4 bits of X) * 256 + Y + 1.

Imploding - Method 6
```

The Imploding algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences using a sliding dictionary. The second algorithm is used to compress the encoding of the sliding dictionary output, using multiple Shannon-Fano trees.

The Imploding algorithm can use a 4K or 8K sliding dictionary size. The dictionary size used can be determined by bit 1 in the general purpose flag word; a 0 bit indicates a 4K dictionary while a 1 bit indicates an 8K dictionary.

The Shannon-Fano trees are stored at the start of the compressed file. The number of trees stored is defined by bit 2 in the general purpose flag word; a 0 bit indicates two trees stored, a 1 bit indicates three trees are stored. If 3 trees are stored, the first Shannon-Fano tree represents the encoding of the Literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the Distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

The Literal Shannon-Fano tree, if present is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is present, the Minimum Match Length for the sliding dictionary is 3. If this tree is not present, the Minimum Match Length is 2.

The Length Shannon-Fano tree is used to compress the Length part of the (length,distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the Minimum Match Length, to 63 plus the Minimum Match Length.

The Distance Shannon-Fano tree is used to compress the Distance part of the (length,distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. The first byte of the tree data represents the number of bytes of data representing the (compressed) Shannon-Fano tree minus 1. The remaining bytes represent the Shannon-Fano tree data encoded as:

```
High 4 bits: Number of values at this bit length + 1. (1 - 16)
Low  4 bits: Bit Length needed to represent value + 1. (1 - 16)
```

The Shannon-Fano codes can be constructed from the bit lengths using the following algorithm:

1) Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.

2) Generate the Shannon-Fano trees:

```
Code <- 0
CodeIncrement <- 0
LastBitLength <- 0
i <- number of Shannon-Fano codes - 1   (either 255 or 63)
```

```
loop while i >= 0
    Code = Code + CodeIncrement
    if BitLength(i) <> LastBitLength then
        LastBitLength=BitLength(i)
        CodeIncrement = 1 shifted left (16 - LastBitLength)
    ShannonCode(i) = Code
    i <- i - 1
end loop
```

3) Reverse the order of all the bits in the above ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would become 0x2C48 (hex).

4) Restore the order of Shannon-Fano codes as originally stored within the file.

Example:

This example will show the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

Example: 0x02, 0x42, 0x01, 0x13

The first byte indicates 3 values in this table. Decoding the bytes:

```
0x42 = 5 codes of 3 bits long
0x01 = 1 code  of 2 bits long
0x13 = 2 codes of 4 bits long
```

This would generate the original bit length array of:
(3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 thru 7. Using the algorithm to obtain the Shannon-Fano codes produces:

Val	Sorted	Constructed Code	Reversed Value	Order Restored	Original Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

The values in the Val, Order Restored and Original Length columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. How to parse the variable length Shannon-Fano values from the data stream is beyond the scope of this document. (See the references listed at the end of this document for more information.) However, traditional decoding schemes used for Huffman variable length decoding, such as the Greenlaw algorithm, can be successfully applied.

The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted as follows:

```
loop until done
    read 1 bit from input stream.

    if this bit is non-zero then       (encoded data is literal data)
        if Literal Shannon-Fano tree is present
            read and decode character using Literal Shannon-Fano tree.
        otherwise
            read 8 bits from input stream.
        copy character to the output stream.
    otherwise              (encoded data is sliding dictionary match)
        if 8K dictionary size
            read 7 bits for offset Distance (lower 7 bits of offset).
        otherwise
            read 6 bits for offset Distance (lower 6 bits of offset).

        using the Distance Shannon-Fano tree, read and decode the
          upper 6 bits of the Distance value.

        using the Length Shannon-Fano tree, read and decode
          the Length value.

        Length <- Length + Minimum Match Length

        if Length = 63 + Minimum Match Length
            read 8 bits from the input stream,
            add this value to Length.

        move backwards Distance+1 bytes in the output stream, and
        copy Length characters from this position to the output
        stream.  (if this position is before the start of the output
        stream, then assume that all the data before the start of
        the output stream is filled with zeros).
end loop
```

Tokenizing - Method 7

This method is not used by PKZIP.

Deflating - Method 8

The Deflate algorithm is similar to the Implode algorithm using a sliding dictionary of up to 32K with secondary compression from Huffman/Shannon-Fano codes.

The compressed data is stored in blocks with a header describing the block and the Huffman codes used in the data block. The header format is as follows:

```
   Bit 0: Last Block bit     This bit is set to 1 if this is the last
                             compressed block in the data.
   Bits 1-2: Block type
      00 (0) - Block is stored - All stored data is byte aligned.
               Skip bits until next byte, then next word = block
               length, followed by the ones compliment of the block
               length word. Remaining data in block is the stored
               data.

      01 (1) - Use fixed Huffman codes for literal and distance codes.
               Lit Code    Bits             Dist Code   Bits
               ---------   ----             ---------   ----
                 0 - 143    8                 0 - 31      5
```

```
          144 - 255     9
          256 - 279     7
          280 - 287     8

          Literal codes 286-287 and distance codes 30-31 are
          never used but participate in the huffman construction.

      10 (2) - Dynamic Huffman codes.  (See expanding Huffman codes)

      11 (3) - Reserved - Flag a "Error in compressed data" if seen.
```

Expanding Huffman Codes

If the data block is stored with dynamic Huffman codes, the Huffman codes are sent in the following compressed format:

```
   5 Bits: # of Literal codes sent - 256 (256 - 286)
           All other codes are never sent.
   5 Bits: # of Dist codes - 1           (1 - 32)
   4 Bits: # of Bit Length codes - 3     (3 - 19)
```

The Huffman codes are sent as bit lengths and the codes are built as described in the implode algorithm. The bit lengths themselves are compressed with Huffman codes. There are 19 bit length codes:

```
   0 - 15: Represent bit lengths of 0 - 15
       16: Copy the previous bit length 3 - 6 times.
           The next 2 bits indicate repeat length (0 = 3, ... ,3 = 6)
              Example:  Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will
                        expand to 12 bit lengths of 8 (1 + 6 + 5)
       17: Repeat a bit length of 0 for 3 - 10 times. (3 bits of length)
       18: Repeat a bit length of 0 for 11 - 138 times (7 bits of length)
```

The lengths of the bit length codes are sent packed 3 bits per value (0 - 7) in the following order:

```
   16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
```

The Huffman codes should be built as described in the Implode algorithm except codes are assigned starting at the shortest bit length, i.e. the shortest code should be all 0's rather than all 1's. Also, codes with a bit length of zero do not participate in the tree construction. The codes are then used to decode the bit lengths for the literal and distance tables.

The bit lengths for the literal tables are sent first with the number of entries sent described by the 5 bits sent earlier. There are up to 286 literal characters; the first 256 represent the respective 8 bit character, code 256 represents the End-of-Block code, the remaining 29 codes represent copy lengths of 3 thru 258. There are up to 30 distance codes representing distances from 1 thru 32k as described below.

Length Codes

Code	Extra Bits	Length	Code	Extra Bits	Lengths	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	265	1	11,12	273	3	35-42	281	5	131-162
258	0	4	266	1	13,14	274	3	43-50	282	5	163-194
259	0	5	267	1	15,16	275	3	51-58	283	5	195-226
260	0	6	268	1	17,18	276	3	59-66	284	5	227-257
261	0	7	269	2	19-22	277	4	67-82	285	0	258

262	0	8	270	2	23-26	278	4	83-98
263	0	9	271	2	27-30	279	4	99-114
264	0	10	272	2	31-34	280	4	115-130

Distance Codes

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance	Code	Extra Bits	Distance
0	0	1	8	3	17-24	16	7	257-384	24	11	4097-6144
1	0	2	9	3	25-32	17	7	385-512	25	11	6145-8192
2	0	3	10	4	33-48	18	8	513-768	26	12	8193-12288
3	0	4	11	4	49-64	19	8	769-1024	27	12	12289-16384
4	1	5,6	12	5	65-96	20	9	1025-1536	28	13	16385-24576
5	1	7,8	13	5	97-128	21	9	1537-2048	29	13	24577-32768
6	2	9-12	14	6	129-192	22	10	2049-3072			
7	2	13-16	15	6	193-256	23	10	3073-4096			

The compressed data stream begins immediately after the compressed header data. The compressed data stream can be interpreted as follows:

```
do
   read header from input stream.

   if stored block
      skip bits until byte aligned
      read count and 1's compliment of count
      copy count bytes data block
   otherwise
      loop until end of block code sent
         decode literal character from input stream
         if literal < 256
            copy character to the output stream
         otherwise
            if literal = end of block
               break from loop
            otherwise
               decode distance from input stream

               move backwards distance bytes in the output stream, and
               copy length characters from this position to the output
               stream.
      end loop
while not last block

if data descriptor exists
   skip bits until byte aligned
   read crc and sizes
endif
```

Enhanced Deflating - Method 9

The Enhanced Deflating algorithm is similar to Deflate but uses a sliding dictionary of up to 64K. Deflate64(tm) is supported by the Deflate extractor.

BZIP2 - Method 12

BZIP2 is an open-source data compression algorithm developed by Julian Seward. Information and source code for this algorithm

can be found on the internet.

Traditional PKWARE Encryption

The following information discusses the decryption steps required to support traditional PKWARE encryption. This form of encryption is considered weak by todays standards and its use is recommended only for situations with low security needs or for compatiblity with older .ZIP applications.

Decryption

The encryption used in PKZIP was generously supplied by Roger Schlafly. PKWARE is grateful to Mr. Schlafly for his expert help and advice in the field of data encryption.

PKZIP encrypts the compressed data stream. Encrypted files must be decrypted before they can be extracted.

Each encrypted file has an extra 12 bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using three, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using pseudo-random number generation techniques in combination with the same CRC-32 algorithm used in PKZIP and described elsewhere in this document.

The following is the basic steps required to decrypt a file:

1) Initialize the three 32-bit keys with the password.
2) Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
3) Read and decrypt the compressed data stream using the encryption keys.

Step 1 - Initializing the encryption keys

```
Key(0) <- 305419896
Key(1) <- 591751049
Key(2) <- 878082192

loop for i <- 0 to length(password)-1
    update_keys(password(i))
end loop
```

Where update_keys() is defined as:

```
update_keys(char):
  Key(0) <- crc32(key(0),char)
  Key(1) <- Key(1) + (Key(0) & 000000ffH)
  Key(1) <- Key(1) * 134775813 + 1
  Key(2) <- crc32(key(2),key(1) >> 24)
end update_keys
```

Where crc32(old_crc,char) is a routine that given a CRC value and a character, returns an updated CRC value after applying the CRC-32 algorithm described elsewhere in this document.

Step 2 - Decrypting the encryption header

The purpose of this step is to further initialize the encryption keys, based on random data, to render a plaintext attack on the data ineffective.

Read the 12-byte encryption header into Buffer, in locations Buffer(0) thru Buffer(11).

```
loop for i <- 0 to 11
    C <- buffer(i) ^ decrypt_byte()
    update_keys(C)
    buffer(i) <- C
end loop
```

Where decrypt_byte() is defined as:

```
unsigned char decrypt_byte()
    local unsigned short temp
    temp <- Key(2) | 2
    decrypt_byte <- (temp * (temp ^ 1)) >> 8
end decrypt_byte
```

After the header is decrypted, the last 1 or 2 bytes in Buffer should be the high-order word/byte of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. Versions of PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is used on versions after 2.0. This can be used to test if the password supplied is correct or not.

Step 3 - Decrypting the compressed data stream

The compressed data stream can be decrypted as follows:

```
loop until done
    read a character into C
    Temp <- C ^ decrypt_byte()
    update_keys(temp)
    output Temp
end loop
```

Strong Encryption (EFS)

Version 5.x of this specification includes support for strong encryption algorithms. These algorithms can be used with either a password or an X.509v3 digital certificate to encrypt each file. This format specification supports either password or certificate based encryption to meet the security needs of today, to enable interoperability between users within both PKI and non-PKI environments, and to ensure interoperability between different computing platforms that are running a ZIP program.

Password based encryption is the most common form of encryption people are familiar with. However, inherent weaknesses with passwords (e.g. susceptibility to dictionary/brute force attack) as well as password management and support issues make certificate based encryption a more secure and scalable option. Industry efforts and support are defining and moving towards more advanced security solutions built around X.509v3 digital certificates and

Public Key Infrastructures(PKI) because of the greater scalability, administrative options, and more robust security over traditional password-based encryption.

Most standard encryption algorithms are supported with this specification. Reference implementations for many of these algorithms are available from either commercial or open source distributors. Readily available cryptographic toolkits make implementation of the encryption features straight-forward.

The algorithms introduced in Version 5.0 of this specificaion include:

    RC2 40 bit, 64 bit, and 128 bit
    RC4 40 bit, 64 bit, and 128 bit
    DES
    3DES 112 bit and 168 bit

Version 5.1 adds support for the following:

    AES 128 bit, 192 bit, and 256 bit

The details of the strong encryption specification for certificates remain under development as design and testing issues are worked out for the range of algorithms, encryption methods, certificate processing and cross-platform support necessary to meet the advanced security needs of .ZIP file users today and in the future.

This feature specification is intended to support basic encryption needs of today, such as password support. However this specification is also designed to lay the foundation for future advanced security needs.

Password-based encryption using strong encryption algorithms operates similarly to the traditional PKWARE encryption defined in this format. Additional data structures are added to support the processing needs of the strong algorithms.

The Strong Encryption data structures are:

1. Bits 0 and 6 of the General Purpose bit flag in both local and central header records. Both bits set indicates strong encryption.

2. Extra Field 0x0017 in central header only.

    Fields to consider in this record are:

    Format - the data format identifier for this record. The only value allowed at this time is the integer value 2.

    AlgId - integer identifier of the encryption algorithm from the following range

        0x6601 - DES
        0x6602 - RC2 (version needed to extract < 5.2)
        0x6603 - 3DES 168
        0x6609 - 3DES 112
        0x660E - AES 128
        0x660F - AES 192
        0x6610 - AES 256

```
    0x6702 - RC2 (version needed to extract >= 5.2)
    0x6801 - RC4
    0xFFFF - Unknown algorithm

Bitlen - Explicit bit length of key

     40
     64
    112
    128
    192
    256

Flags - Processing flags needed for decryption

    0x0001 - Password is required to decrypt
    0x0002 - reserved for certificates only
    0x0003 - Password or certificate required to decrypt

    Values > 0x0003 reserved for certificate processing


3. Decryption header record preceeding compressed file data.

      -Decryption Header:

       Value     Size     Description
       -----     ----     -----------
       IVSize    2 bytes  Size of initialization vector (IV)
       IVData    IVSize   Initialization vector for this file
       Format    2 bytes  Format definition for this record
       AlgID     2 bytes  Encryption algorithm identifier
       Bitlen    2 bytes  Bit length of encryption key
       Flags     2 bytes  Processing flags
       ErdSize   2 bytes  Size of Encrypted Random Data
       ErdData   ErdSize  Encrypted Random Data
       Reserved1 4 bytes  Reserved certificate data
       Reserved2 (var)    Reserved for certificate data
       VSize     2 bytes  Size of password validation data
       VData     VSize-4  Password validation data
       VCRC32    4 bytes  CRC32 of password validation data

   IVData - The size of the IV should match the algorithm block size.
            The IVData can be completely random data.  If the size of
            the randomly generated data does not match the block size
            it should be complemented with zero's.  If IVSize is 0,
            then IV = CRC32 + 64-bit File Size.

   Format - the data format identifier for this record.  The only
   value allowed at this time is the integer value 3.

   AlgId - integer identifier of the encryption algorithm from the
   following range

       0x6601 - DES
       0x6602 - RC2 (version needed to extract < 5.2)
       0x6603 - 3DES 168
       0x6609 - 3DES 112
       0x660E - AES 128
       0x660F - AES 192
       0x6610 - AES 256
       0x6702 - RC2 (version needed to extract >= 5.2)
       0x6801 - RC4
```

```
        0xFFFF - Unknown algorithm

    Bitlen - Explicit bit length of key

         40
         64
        112
        128
        192
        256

    Flags - Processing flags needed for decryption

        0x0001 - Password is required to decrypt
        0x0002 - reserved for certificates only
        0x0003 - Password or certificate required to decrypt

        Values > 0x0003 reserved for certificate processing

    ErdData - Encrypted random data is used to generate a file
              session key for encrypting each file.  SHA1 is
              used to calculate hash data used to derive keys.
              File session keys are deived from a master session
              key generated from the user-supplied password.

    Reserved1 - Reserved for certificate processing, if value is
              zero, then Reserved2 data is absent.

    VSize - This size value will always include the 4 bytes of the
            VCRC32 data and will be greater than 4 bytes.

    VData - Random data for password validation.  This data is VSize
            in length and VSize must be a multiple of the encryption
            block size.  VCRC32 is a checksum value of VData.  VSize,
            VData, and VCRC32 are stored encrypted and start the
            stream of encrypted data for a file.

Strong Encryption is always applied to a file after compression. The
block oriented algorithms all operate in Cypher Block Chaining (CBC)
mode.  The block size used for AES encryption is 16.  All other block
algorithms use a block size of 8.  Two ID's are defined for RC2 to
account for a discrepancy found in the implementation of the RC2
algorithm in the cryptographic library on Windows XP SP1 and all
earlier versions of Windows.

A pseudo-code representation of the encryption process is as follows:

Password = GetUserPassword()
RD  = Random()
ERD = Encrypt(RD,DeriveKey(SHA1(Password)))
For Each File
    IV = Random()
    VData = Random()
    FileSessionKey = DeriveKey(SHA1(RD, IV))
    Encrypt(VData + FileData,FileSessionKey)
Done

The function names and parameter requirements will depend on
the choice of the cryptographic toolkit selected.  Almost any
toolkit supporting the reference implementations for each
algorithm can be used.  The RSA BSAFE(r), OpenSSL, and Microsoft's
CryptoAPI libraries are all known to work well.
```

Change Process

In order for the .ZIP file format to remain a viable definition, this specification should be considered as open for periodic review and revision. Although this format was originally designed with a certain level of extensibility, not all changes in technology (present or future) were or will be necessarily considered in its design. If your application requires new definitions to the extensible sections in this format, or if you would like to submit new data structures, please forward your request to zipformat@pkware.com. All submissions will be reviewed by the ZIP File Specification Committee for possible inclusion into future versions of this specification. Periodic revisions to this specification will be published to ensure interoperability.

Acknowledgements

In addition to the above mentioned contributors to PKZIP and PKUNZIP, I would like to extend special thanks to Robert Mahoney for suggesting the extension .ZIP for this software.

References:

Fiala, Edward R., and Greene, Daniel H., "Data compression with finite windows", Communications of the ACM, Volume 32, Number 4, April 1989, pages 490-505.

Held, Gilbert, "Data Compression, Techniques and Applications, Hardware and Software Considerations", John Wiley & Sons, 1987.

Huffman, D.A., "A method for the construction of minimum-redundancy codes", Proceedings of the IRE, Volume 40, Number 9, September 1952, pages 1098-1101.

Nelson, Mark, "LZW Data Compression", Dr. Dobbs Journal, Volume 14, Number 10, October 1989, pages 29-37.

Nelson, Mark, "The Data Compression Book", M&T Books, 1991.

Storer, James A., "Data Compression, Methods and Theory", Computer Science Press, 1988

Welch, Terry, "A Technique for High-Performance Data Compression", IEEE Computer, Volume 17, Number 6, June 1984, pages 8-19.

Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression", Communications of the ACM, Volume 30, Number 6, June 1987, pages 520-540.

Ziv, J. and Lempel, A., "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, Volume 24, Number 5, September 1978, pages 530-536.